LA-UR- 09-00812

Approved for public release;
distribution is unlimited.

Title:	Dynamic Load Balancing of Matrix-Vector Multiplications on Roadrunner Compute Nodes
Author(s):	Jose Carlos Sancho Pitarch, CCS-1 Darren J. Kerbyson, CCS-1
Intended for:	Euro-Par 2009 Conference Delft, The Netherlands August 25, 2009

Form 836 (7/06)

Dynamic Load Balancing of Matrix-Vector Multiplications on Roadrunner Compute Nodes

José Carlos Sancho and Darren J. Kerbyson

Performance and Architecture Laboratory (PAL),
Los Alamos National Laboratory, NM 87545, USA

Abstract. Hybrid architectures that combine general purpose processors with accelerators are being adopted in several large-scale systems such as the petaflop Roadrunner supercomputer at Los Alamos. In this system, dual-core Opteron host processors are tightly coupled with PowerXCell 8i processors within each compute node. In this kind of hybrid architecture, an accelerated mode of operation is typically used to offload performance hotspots in the computation to the accelerators. In this paper we explore the suitability of a variant of this acceleration mode in which the performance hotspots are actually shared between the host and the accelerators. To achieve this we have designed a new load balancing algorithm, which is optimized for the Roadrunner compute nodes, to dynamically distribute computation and associated data between the host and the accelerators at runtime. Results are presented using this approach for sparse and dense matrix-vector multiplications that show load-balancing can improve performance by up to 24% over solely using the accelerators.

1 Introduction

The unprecedented need for power efficiency is currently driven the design of hybrid computer architectures that combine traditional general purpose processors with specialized high-performance accelerators. Such a hybrid architecture has been recently installed at Los Alamos National Laboratory in the form of the Roadrunner supercomputer [1]. This system was the first to achieve a sustained performance of over 1 *PetaFlop/s* on the LINPACK benchmark.

In Roadrunner, dual-core Opteron host processors are tightly coupled with PowerXCell 8i processors [5] within each compute node. This hybrid architecture can support several types of processing modes including: host-centric, accelerator-centric, and an accelerated mode of operation. The characteristics of an application determines which mode is most suitable. The host-centric mode can be though of as the traditional mode of operation where applications solely use the host Opteron processors. In the accelerator-centric mode applications solely run on the PowerXCell 8i. In the accelerated mode, both Opteron and PowerXCell 8i are used in such a way that performance-critical sections of computation are off-loaded to PowerXCell 8i accelerators leaving the rest of the code

to run on the host Opterons. SPaSM, a molecular dynamics code, is an example of an application that followed this accelerated approach [10].

A variant of the accelerated mode is to share the performance hotspots between both the accelerator and host processors for simultaneous processing. The benefit of this is a potential gain in performance, since the computation power of both the host processors and accelerators can be harnessed simultaneously, but with an associated increase in complexity. The computation power of the host processors may be orders of magnitude smaller than that of the accelerators but at large-scale, including Roadrunner, the performance gain can be significant and thus should be exploited. However, this kind of accelerated mode increases complexity - extra tools are required in order efficiently and dynamically load balance between the hosts and accelerators at runtime. Undertaking such a load-balance during application execution is desirable in this context as it is difficult to determine costs associated with individual computations at compile-time, and there may be changes in the amount of data to compute per processor during runtime which can result in repartitioning across nodes.

This paper addresses this challenge and presents a load balancing algorithm in order to dynamically distribute the computation and associated data between the host Opterons and the PowerXCell 8i accelerators at runtime in the compute nodes of Roadrunner. For illustration purposes we address the common operation of matrix-vector multiplications on the form of $y = y + Ax$, where A is either a sparse or dense matrix and x and y are dense vectors. These operations are commonly found in scientific applications and are prime candidates to offload to accelerators. Results show that the dynamic load balancing algorithm can improve the performance of these operations by up to 24% when using both host and accelerator processors in comparison to solely using the accelerators. In addition, the determination of the optimal load balance converges quickly taking only 7 iterations. Quick convergence is desirable for a dynamic load balancing algorithm in order to minimize its impact on the overall runtime. Although the results as presented consider one compute node of Roadrunner, there is nothing to prevent our technique to be applied to larger-node counts up to a system-wide parallel job.

The rest of this paper is organized as follows. Section 2 describes the architecture of a Roadrunner compute node. Section 3 describes our load balancing algorithm. Section 4 briefly describes the implementation of matrix-vector multiplications on the PowerXCell 8i and includes experimental results from a Roadrunner node. Related work on matrix vector multiplications in hybrid architectures is summarized in Section 5. And finally, conclusions from this work are given in Section 6.

2 The Roadrunner Compute Node

A compute node of Roadrunner is built using three compute blades and one interconnect blade as shown in Figure 1. A single IBM LS21 blade contains two 1.8GHz dual-core Opteron processors, and two IBM QS22 blades each contain

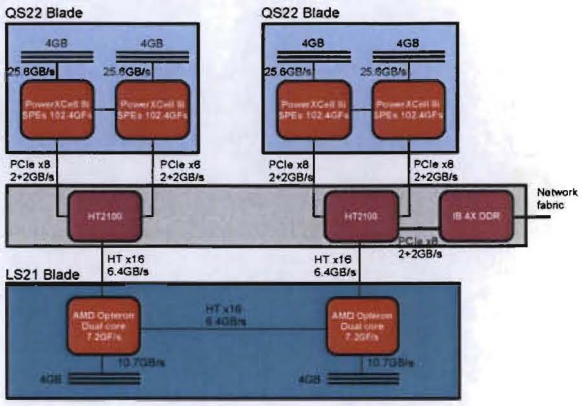

Fig. 1. The structure of a Roadrunner compute node.

two 3.2GHz PowerXCell 8i processors [5]. The fourth blade interconnects the three compute blades using two Broadcom HT2100 I/O controllers. These controllers convert the HyperTransport 16x connections from the Opterons to PCIe x8 buses — one to each PowerXCell 8i. In this configuration each Opteron core is uniquely paired with a PowerXCell 8i processor when using the accelerated mode of operation.

The PowerXCell 8i processors have approximately 95% of the peak floating-point performance and 80% of the peak memory bandwidth of a node respectively. Each PowerXCell 8i consists of eight Synergistic Processing Elements (SPEs) and one Power Processing Element (PPE). The eight SPEs have an aggregate peak performance of 102.4 GFlops/s (double-precision), or 204.8 Flops/s (single-precision) whereas dual-core Opteron has a peak performance of 7.2 GFlops/s (double-precision) or 14.4GFlops/s (single-precision). Therefore, the PowerXCell 8i can potentially accelerate a compute-bound code by up to 28× (102.4/3.6) over a single Opteron core. In addition, each PowerXCell 8i processor has substantially more memory bandwidth than the Opterons, 25.6 GB/s compared to 10.7GB/s for a dual-core Opteron.

The PPE is a PowerPC processor core which runs the OS and manages the SPEs. The SPEs are in-order execution processors with a two-way SIMD that do not have a cache. Instead they can directly access a 256KB high-speed memory called a *local store* which is explicitly accessed by direct memory access (DMA) transfers from the PPE memory space. Each compute node has a total of 32GB of memory evenly distributed across the six processors (Opteron and PowerXCell 8i) providing each with 4 GB of memory.

3 The Dynamic Load Balance Algorithm

In this section we describe our dynamic load balancing algorithm applied to matrix-vector multiplication. These operations are very time-consuming in codes

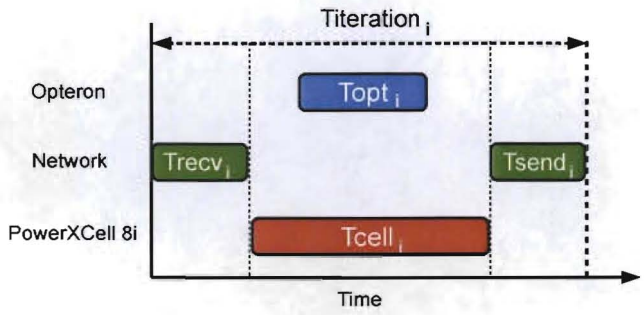

Fig. 2. Breakdown of iteration time (non-overlapping transfers).

such as iterative solvers where the matrix-vector multiplication, $y_{i+1} = y_i + Ax$, is performed once or more in each iteration of the application. We chose a single row of the matrix A as the smallest granularity of load-balancing the data between the Opteron and PowerXCell 8i. The goal of the load-balancing algorithm is to find an optimal partitioning of the matrix rows to minimize the runtime when this calculation is performed multiple times. Formally, the load-balancing problem can be described as the following optimization problem:

$$Truntime = min \sum_{i=1}^{n} (max(Topt_i + Transfer_i, Tcell_i + Ttransfer_i) + Thousekeeping_i)$$

where $Truntime$ is the total execution time of the matrix-vector multiplication for n iterations; $Topt_i$ and $Tcell_i$ are the times to perform the associated matrix-vector multiplications on the Opteron and PowerXCell 8i for iteration i; $Ttransfer_i$ is the sum of times for receiving data to compute on the PowerXCell 8i ($Trecv_i$) and for sending back the results ($Tsend_i$) to the Opteron; and finally, $Thousekeeping_i$ is the time associated for the load-balancing algorithm and formatting the data for processing on the PowerXCell 8i.

We follow the operation of iterative solvers where the data that is transferred in and out of the operation in each iteration are the vectors y_i and y_{i+1} respectively. The matrix A is considered constant as in most iterative solvers, and hence it does not need to be transferred to the PowerXCell 8i each iteration. Similarly, the vector x also does not need to the transferred each iteration as it is computed internally based on the residuals. Notwithstanding, there are applications that the matrix A actually does change per iteration such as in the case of *Adaptive Mesh Refinement* codes. For such codes, the dynamic load-balancing algorithm would be re-run again in order to find the best load-balance for the new matrix A when it is a sparse matrix. Usually, the strategy taken is to wait until the number of changes in the matrix A are larger than a given threshold in order to minimize the *housekeeping* costs associated with load-balancing.

From the point of view of one iteration, the optimization problem is reduced to the case illustrated in Figure 2 that shows the elapsed times on the Opteron,

PowerXCell 8i, and the intranode-connection network in the case that the data transfers are not being overlapped with the computation. When the transfers are not overlapped, there is an additional cost for both the Opteron and PowerXCell 8i that needs to be taken into account in the minimization problem. Therefore, we want to minimize both $Topt + Ttransfer$ and $Tcell + Ttransfer$ at the same time, i.e. minimize $max(Topt + Ttransfer, Tcell + Ttransfer)$. By distributing the data carefully between processors it is possible to achieve the optimal balance that minimizes the above expression. For example, in the case that the PowerXCell 8i has to much to compute, we can move some of data to the Opteron which reduces both $Tcell$ and $Ttransfer$ at expenses of increasing $Topt$. Careful attention should be taken to prevent the case that the Opteron has too much data to compute, $Topt > Tcell$, which will also increase the iteration time. In the converse case, that the Opteron has too much data, some data can be moved from the Opteron to the PowerXCell 8i. Note again that assigning more data to the PowerXCell 8i in the next iteration, $i+1$, means that both the $Tcell_{i+1}$ and $Ttransfer_{i+1}$ will be increased. And therefore, the iteration time may be larger because the $Ttransfer_{i+1}$ might be too high to offset the reduction in time on the Opteron, $Topt_{i+1} + Ttransfer_{i+1} > Topt_i + Ttransfer_i$. It can also occur that the cell has to much data to compute with respect to the Opteron, $Tcell_{i+1} > Topt_{i+1}$.

When the data transfers can be fully overlapped with computation, the load-balancing is simplified to the case of making the compute-times on both the Opteron and PowerXCell 8i equal, $Topt \simeq Tcell$, in order to minimize the following expression $max(Topt, Tcell)$. This is an ideal case that might be difficult to achieve in a real scenario because it depends on the application's data dependencies— data is not available yet because it needs to be combined with other data such as in the case of iterative solvers—, and the support of asynchronous operations on the communication system. Although, Roadrunner supports asynchronous communications, the data dependencies can prevent fully overlapped operation. Hence; the common scenario is that communications are only partially overlapped and the optimization problem described in Figure 2 applies.

In addition, the housekeeping cost of the load balancing algorithm is only paid during the time to converge. Once the algorithm has converged the data structures are set up in the optimal load-balance configuration, and thus no further load-balancing is required. Therefore, it is desirable to converge as fast as possible in order to minimize the impact of the housekeeping costs on the runtime of the application.

The load-balancing algorithm proposed is based on combining the following three basic approaches: accelerator-centric, performance-based, and trial-and-error in order to converge at the optimal state as quickly as possible. This algorithm is comprised of six states as depicted in Figure 3. For the sake of this description, *ratio* is defined as the percent of data that is assigned to the Opteron per iteration. In the first state, we take an accelerator-centric approach where *ratio* is initialized to be *Peak cell/Peak opt*, where *Peak cell* and *Peak opt* are

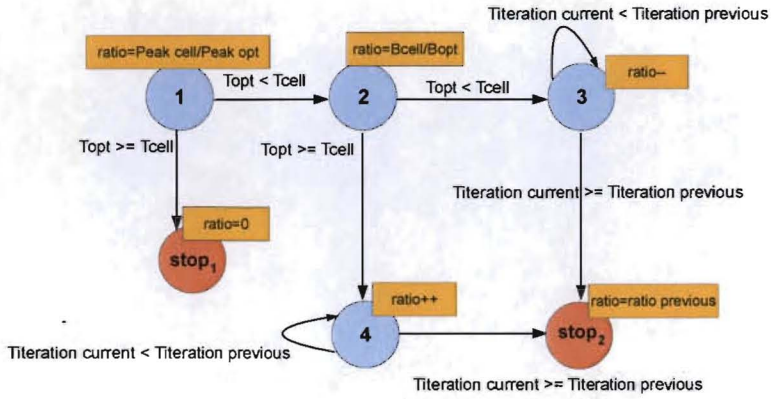

Fig. 3. States of the load balance algorithm.

the peak flop performance of the PowerXCell 8i and Opteron, respectively. In Roadrunner, *ratio* is initialized to be 28, see Section 2. We use the peak performance of the processors as an starting point as this is available a priori. In principle, we do not know anything about the characteristics of the code and the peak flop performance is a save alternative in this architecture with respect to the peak memory bandwidth because most of the work will be performed on the PowerXCell 8i rather than the Opteron.

In the second state, we take a performance-based approach since we can collect actual timing information. The principle of a performance-based approach is to distribute data based on how well the different processors perform, and thus allowing the algorithm to quickly converge to the optimal ratio. This is achieved by collecting the times, $Topt$ and $Tcell$, in order to calculate the processing rates, $Bcell$ and $Bopt$, for both the Opteron and PowerXCell 8i.

Note that $Tcell$ and $Ttransfer$ are measured independently instead of combining them into a single metric. This distinction is more efficient than the typical combination approach of as will be shown in the next section. In this state, when $Topt >= Tcell$ then the Opteron has been assigned too much data. However, the *ratio* was set up already small, and hence, those processors should no longer be considered and the load-balancing is placed in the $stop_1$ state with $ratio = 0$. When $Topt < Tcell$ then the Opteron has capability to undertake more work and thus the *ratio* is set based on the current measured processing rate of the processors ($Bcell/Bopt$). Additionally, we measure the execution time per iteration ($Titeration$) which takes into account the $Ttransfer$ for the next iteration of the load-balancing algorithm.

Finally, the third and fourth steps are performed using a trial-and-error load-balancing strategy until the optimal balance is achieved. This is done by carefully assigning more or less data on the Opteron in order to not increase the $Titeration$. Note that these additional steps are not included in a typical performance-based load balancing strategy, but they were necessary for the case of this particular architecture. In particular, the third step gradually decreases

ratio in the case that $Topt < Tcell$, so assigning more data to the Opteron. Similarly, the fourth step gradually increases *ratio* in the case that $Topt >= Tcell$ assigning less data to the Opteron. Convergence is achieved when the current $Titeration$ is higher than the previous $Titeration$ time stopping the algorithm in state $stop_2$. Note again, that for the case of fully overlapping transfers these additional states might not lead to the optimal balance as state 2 should already give a good balance due to the fact that it is based on the achieved processing rate and the transfer time does not impact on the iteration time. However, these states are necessary in the case of partially overlapping and non-overlapping transfers where the transfer time does actually impact the iteration time.

This algorithm executes on the Opteron cores per each of the four host-accelerator pairs available in the Roadrunner nodes following a typical global centralized strategy [13]. In this scheme results from the load-balancing are sent to the other processors for use in their data distribution in the next application iteration.

4 Evaluation

We evaluate our load-balancing technique on a Roadrunner compute node as described in Section 2. A four-process parallel job was executed in the accelerated mode of operation that performed a matrix-vector multiplication several times. At the end of the calculation all the processes synchronize in order to account for the worst time. Timing data presented below are averages over multiple runs. We use the DaCS communication library [3] for communicating between Opteron and PowerXCell 8i processors, and OpenMPI version 1.3b [8] message passing library for the synchronization across Opterons. The Cell BE SDK version 3.1 was used to compile the code for the PowerXCell 8i processors.

We evaluated the performance of our load-balance technique, *Optimized balance*, as well as for the case using our load-balance algorithm but considering $Ttransfer$ in combination with $Tcell$, *Balance Ttransfer*. Also for comparison purposes we evaluated the performance of using no load balancing in two cases: using only Opterons and using only PowerXCell 8i processors. In addition, for illustration purposes we show results for a *Greedy* strategy that searches for the optimal load-balance by exploring a wide range of *ratios*: it starts with the default *ratio* (*Peak cell/Peak opt*) and gradually decrements it every iteration to when all work is performed by the Opteron. The experiments were conducted on a dense matrix and on seven sparse matrices from a wide variety of actual applications as listed in Table 1.

The calculation of the matrix-vector multiplications for dense matrices follows a straightforward implementation on the PowerXCell 8i that directly streams the vectors x and y and the matrix A for computing on the SPEs. Due to the limited size of the local-store we need to transfer data into each SPE as needed using DMAs. In the case of matrix A, we assume that each DMA transfer contains one or more entire rows when using a single DMA of 16KB maximum size. Also, matrix A is evenly partitioned among the eight SPEs in each PowerX-

Table 1. Description of the matrices used in the evaluation.

Name	Dimensions	Non-zeros	Description
Dense matrix	2K×2K	4M	Regular dense matrix
Sparse Harbor	47K×47K	2.37M	3D CFD of Charleston harbor
Sparse Dense	2K×2K	4M	Dense sparse matrix
Sparse Fluid	20.7K×20.7K	1.41M	Fluid structure interaction turbulence
Sparse QCD	49K×49K	1.90M	Quark propagators (QCD/LGT)
Sparse Ship	141K×141K	3.98M	FEM ship section/detail production
Sparse Cantiveler	62K×62K	4M	FEM cantiveler
Sparse Spheres	83K×83K	6M	FEM concentric spheres

Cell 8i following a *row partitioning* scheme. This partitioning evenly distributes consecutive rows across the SPEs.

In contrast, the implementation for sparse matrices is a little more complex because it has to deal with the irregularity of the memory access due to the sparsity of the data in the matrix. We used the *Compressed Storage Row* (CSR) format [2] for defining the sparse matrix (A). The CSR format basically uses two data structures, the *row pointer* to index the start of each row with the non-zero elements in A, and the *column pointer* to index the column each element is associated with. Note that both the row and column pointers for the PowerXCell 8i implementation have to be properly re-encoded to be SPE local-store relative. Due to the sparsity of A we cannot use a regular DMA transfer to bring the corresponding x vector elements into local-store. Rather we use a special DMA transfer, *get list*, which gathers independent x elements from main memory and packs them contiguously in the local-store. Because every DMA's source addresses must be 16-byte aligned, some additional padding may be needed if the source address differs from the one required. Note that we only transfer to the local-store unique x elements required for each DMA of A obviating in this way any repetition of these elements among the various rows fitted in a DMA. The prepocessing described above is a part of the *housekeeping* in the load-balancing algorithm.

4.1 Results

Figure 4 shows the iteration time for the *Greedy*, *Optimized balance*, and the *Balance Ttransfer* techniques on the sparse matrix Harbor. As can be seen, the minimum execution time is found at iteration 24 for the *Greedy* technique, where $ratio = 5$. At this point the optimal load balance is achieved and the execution time is improved by 15% with respect to using the default *ratio* ($ratio = Bcell/Bopt$), and 3.4× with respect to $ratio = 1$ (iteration number 28) where all the work is performed by the Opterons. The *Greedy* technique can easily find the optimal balance, but at the expense of a longer converge time (28 iterations) which is undesirable. In contrast, the *Optimized balance* technique converges faster and is able to find the optimal balance after only 5 iterations. Converging faster is desirable as there is extra overhead due to housekeeping per

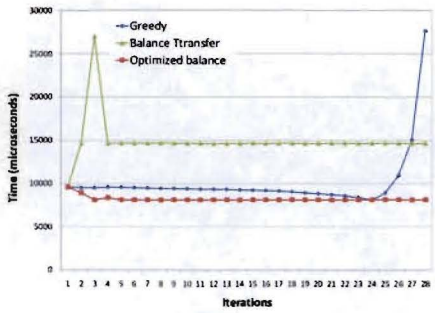

Fig. 4. Iteration time for the Greedy, Optimized balance, and Balance Ttransfer techniques on the sparse matrix Harbor.

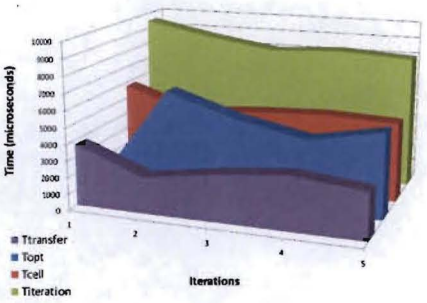

Fig. 5. Iteration time breakdown for the Optimized balance technique on the sparse matrix Harbor (first five iterations).

iteration which could be significant, see Section 3. In the case of the Roadrunner compute node this time is around 60ms per iteration. For the case of the *Balance Ttransfer* technique we can see that the load-balance algorithm does not converge to the optimal solution. This is because including the $Ttransfer$ in the $Tcell$ makes the $ratio$ too low ($ratio = 2$) for this architecture due to $Ttransfer$ being high. This forces the search to stop too early, in state 3 of the algorithm, as the next ratio tried unfortunately does not use the accelerators at all ($ratio = 1$).

Figure 5 illustrates how the *Optimized balance* technique converges to the optimal balance during the first 5 iterations by showing the corresponding times $Topt$, $Tcell$, $Ttransfer$, and $Titeration$ for each iteration. On the first iteration, $Topt$ is too small compared with the $Tcell$ because the default $ratio$ yields too little work for the Opterons compared with the PowerXCell 8i. On the second iteration, the $ratio$ is already fixed to the current performance of the processors ($Bcell/Bopt = 4$), but actually results in too much work for the Opterons. On the third and fourth iterations, the load-balance algorithm is in state 4 increasing the $ratio$ in order to gradually reduce the work on the Opterons. During this, it is found that the third iteration results in a better $Titeration$ time with $ratio = 5$ than the fourth iteration, and so the algorithm stops on the fifth iteration taking the tested best $ratio$ ($ratio = 5$) for subsequent iterations. At the optimal balance, 41% of the iteration time is spent on the $Topt$, and 38% and 22% is spent for the $Tcell$ and $Ttransfer$, respectively.

Figure 6 summarizes the execution iteration time for the suite of matrices evaluated when using *Optimized balance* (once the algorithm converged), when using the Opteron only and when using the PowerXCell 8i only. As can be seen, the *Optimized balance* achieves the best runtimes for all the matrices evaluated. In particular, for the dense matrix the performance improvement is 14% for the *Optimized balance* in comparison to using only the PowerXCell 8i. For the

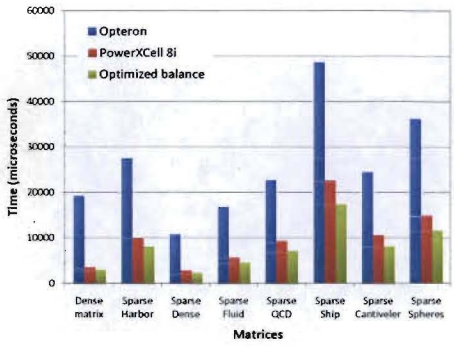

Fig. 6. Execution time for each matrix when using: only the Opteron, only the PowerXCell 8i, and the Optimized balancing technique.

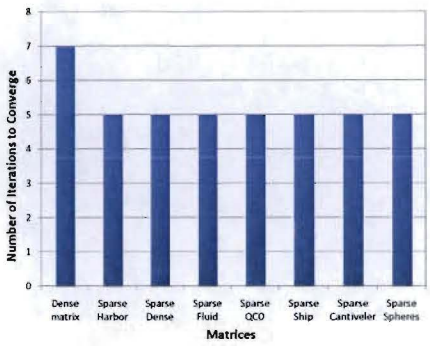

Fig. 7. Number of iterations required for convergence when using the Optimized balancing technique for the testbed matrices.

sparse matrices the improvements are 19%, 18%, 19%, 23%, 23%, 24%, 22% for the sparse matrices Harbor, Dense, Fluid, QCD, Ship, Cantiveler, and Spheres respectively. These improvements are mostly due to the fact that the computation of the sparse matrices is actually memory bound and thus take advantage of the relatively better memory performance of the Opterons rather than their flop performance. As expected, the improvements with respect to the Opteron are more noticeable, ranging from 4× on the sparse Fluid up to 6× for the dense matrix. Also, the number of iterations for the load-balancing to converge for these matrices is small as shown in Figure 7. For the sparse matrices 5 iterations are required for convergence whereas the dense matrix required 7 iterations.

An additional result (not shown in this paper), is the application of our load-balancing technique to the STREAMS microbenchmark [6] that does vector operations instead of matrix-vector operations. Specifically, STREAMS calculates the following $y = y + c \times x$, where y and x are dense vectors and c is a constant. The results of this type of computation showed that the load balancing technique is giving an additional 10% performance increase in comparison to using only the PowerXCell 8i accelerators. For interested readers, the total aggregate memory bandwidth achieved on a Roadrunner compute node using our load balance algorithm was 89GB/s.

5 Related work

Matrix operations including sparse matrix-vector multiplications (SpMV) are key computational kernels in many scientific applications, and thus have been extensively studied. Today most work is focused on implementing these operations on emerging architectures including the Cell BE [12], FPGAs [7], and GPUs [4], as well as multi-core processors [12]. Although our SpMV implementation might not be so highly tuned for a particular processor in comparison to

other implementations, they could be incorporated into our accelerator and host load-balancing method in order to improve overall performance.

On the other hand, there has been very little work on load-balancing matrix operations on hybrid (host-accelerator) architectures since typically they are fully offloaded to the accelerators. However, there is a significant work on load-balancing matrix operations like the SpMV on heterogeneous network of workstations (HNOWs) [13, 9, 11]. These systems are composed of non-uniform processors, network, and memory resources which partially resemble the hybrid platform studied in this work. For HNOWs most of the algorithms are optimized based on the characteristics of the target system. In fact as stated in [13] there is not a unique general solution for all platforms but rather different schemes are best for different applications and system configurations. This result is interesting because it suggests that there should be an efficient load balancing technique as well for our target platform. In particular, our platform is quite different from HNOWs. The processors are tightly attached to each other, so communications are much faster than in HNOWs. Also, there is a huge difference in the computing power of the processor types. These two features open new considerations in the design of load-balancing algorithms that they were not previously important. For example, in this new environment with fast communications it makes more sense to explore fine-grain load balancing algorithms, such as the one proposed in this paper, based on a trial-and-error strategies.

Additionally, in most of the load-balancing strategies for HNOWs distributing the load in proportion to the computing speed of the processors always leads to a perfectly balanced distribution [13, 9]. However, we found that this strategy was not enough to achieve an optimal solution for our platform. In summary, our work represents a step ahead in load-balancing algorithms which is particularly targeted to the hybrid, host-accelerator, architecture of Roadrunner. Notwithstanding, it would be interesting to evaluate as a future work the suitability of our proposed load-balancing algorithm to other hybrid platforms.

6 Conclusions

An optimized load balancing algorithm has been presented in this paper to substantially increase the performance of a Roadrunner compute node. We have demonstrated that the proposed load-balance algorithm achieves a significant performance improvement, up to 24%, when simultaneously using both host (Opteron) and accelerator (PowerXCell 8i) processors in comparison to solely using the PowerXCell 8i processors in a traditional accelerated mode of operation. The load-balancing was evaluated for matrix-vector multiplications which are commonly found in scientific applications, but other operations, including the STREAMS benchmark, have also showed a significant performance improvement of up to 10%.

These improvements come from the concurrent exploitation of the computation power of the host Opteron processors at the same time as the PowerXCell 8i accelerators for processing hotspot computations rather than uniquely offloading

to the accelerators. These results suggest that the traditional accelerated mode of operation is not efficient enough to exploit the full potential of the hybrid architectures including Roadrunner. With effective load-balancing techniques a more complex, but better accelerated mode of operation, can be enabled exploiting concurrently the full potential of all the available processors. In addition, the load-balance algorithm was carefully optimized to provide fast convergence time (7 iterations) making it sufficiently efficient to run during the execution of an application. This feature is desirable in order to dynamically adapt to the characteristics of the code, and thus it can potentially server as a general load-balancing algorithm on this platform for other hotspot computations.

References

1. K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, Entering the Petaflop Era: The Architecture and Performance of Roadrunner, Supercomputing Conference (SC08), Austin, TX, November, 2008.
2. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, Philadelphia, 1994.
3. Data Communication and Synchronization Library for Hybrid-x86: Programmer's Guide and API Reference. IBM Technical document SC33-8408-00. IBM SDK for Multicore Acceleration version 3, release 0. 2007.
4. M. Garland, Sparse matrix computations on manycore GPU's, Annual ACM IEEE Design Automation Conference (DAC08), pp. 2-6, Anaheim, CA, June, 2008.
5. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, Introduction to the Cell Multiprocessor, IBM Journal of Research and Development, 49(4), pp. 589-604, 2005.
6. J. McCalpin, Memory Bandwitdh and Machine Balance in Current High Performance Computers, IEEE Computer Society Technology committee on Computer Architecture (TCCA) Newsletter, pages 19-25, December 1995.
7. G. R. Morris and V. K. Prasanna, Sparse Matrix Computations on Reconfigurable Hardware, IEEE Computer, 40(3), pp. 58-64, 2007.
8. Open-MPI, http://www.open-mpi.org
9. J. F. Pineau, Y. Robert, and F. Vivien, Revisiting matrix product on master-worker platforms, Workshop on Advances in Parallel and Distributed Computational Models (APDCM07), Long Beach, CA, March, 2007.
10. S. Swaminarayan, K. Kadau, and T. C. Germanm. 350-450 Tflops Molecular Dynamics Simulations on the Roadrunner General-purpose Heterogeneous Supercomputer. ACM Gordon Bell Prize finalist, Supercomputing Conference (SC08). Austin, TX, November, 2008.
11. C. Xu and F. Lau, Load Balancing in Parallel Computers: Theory and Practice, Kluwer Academic Publishers, 1996.
12. S. Williams, L. Oliker, R. Vuduc, J. Demmel, and K. Yelick, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, Supercomputing Conference (SC07), Reno, NV, November, 2007.
13. M. J. Zaki, W. Li, and S. Parthasarathy, Customized Dynamic Load Balancing for a Network of Workstations, Parallel and Distributed Computing, 43(2), pp. 156-162, 1997.